devices / FAT

last time (1)

LRU, and (impractically) implementing it

referenced/accessed and dirty bits

second chance:

- ordered list of pages
- take from bottom
- evict if page never referenced while on list
- otherwise return to top, mark unreferenced

SEQ:

- "inactive list" of pages that might be unused
- only check if pages are referenced while on inactive list
- control inactive list size to manage overhead/sensitivity to usage

last time (2)

CLOCK: general idea of scanning/clearing referenced bits periodically

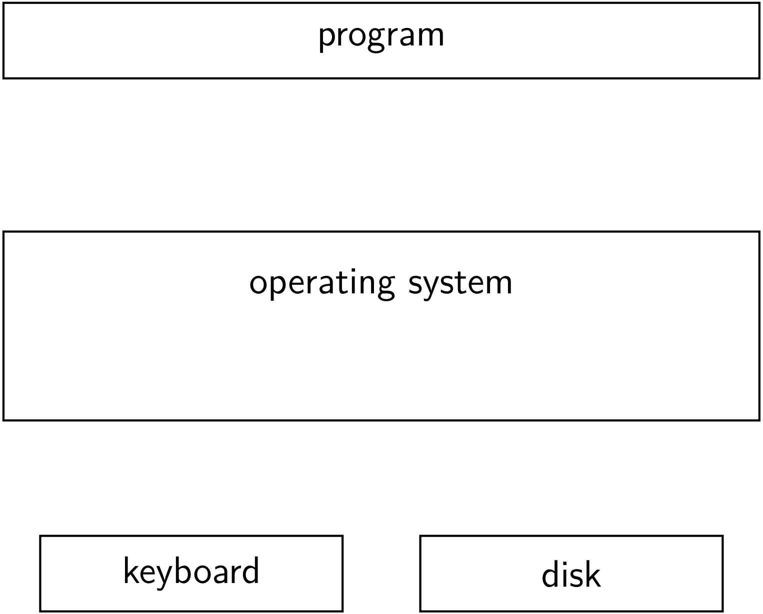
- can be seen as generalization of second chance/SEQ
- variety of LRU-like policies possible

special cases for scanning patterns

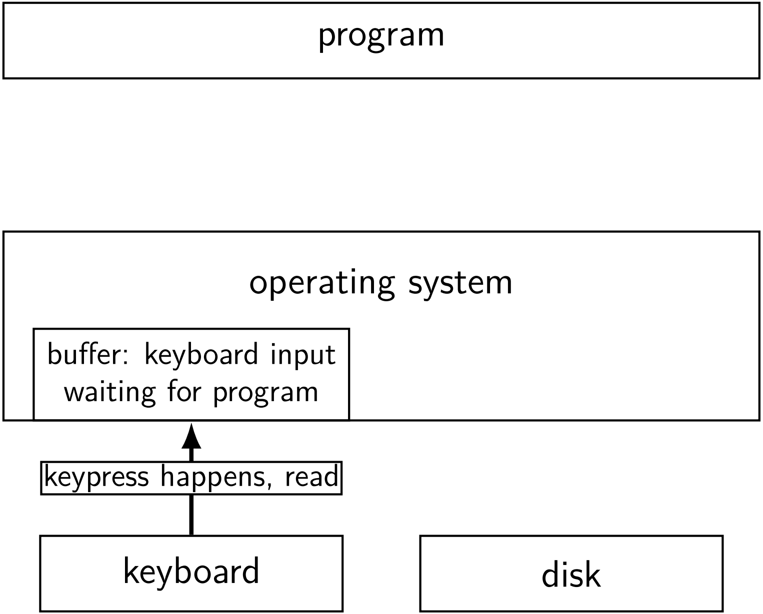
- proactively read in next thing that would be scanned
- maybe don't do LRU-like policy (often not reused)

proactive writing back dirty pages, freeing pages

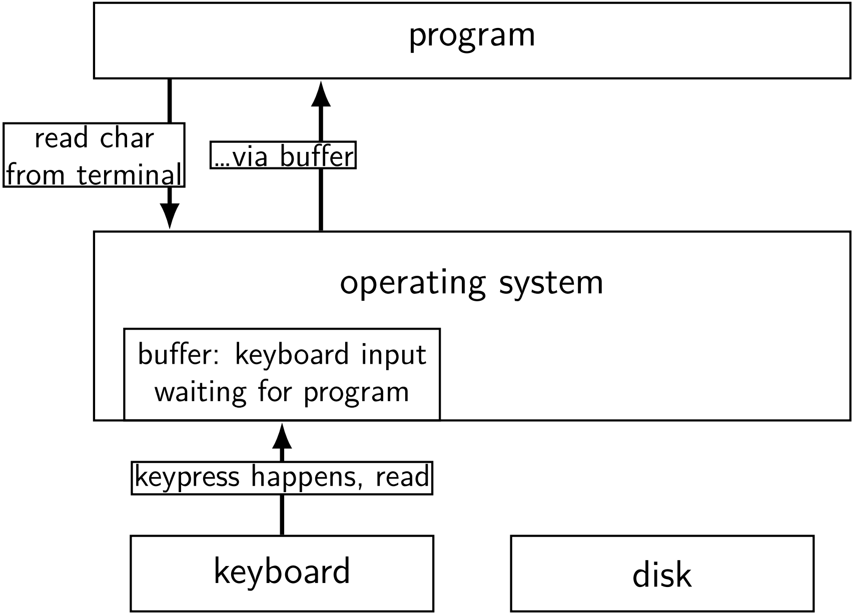
kernel buffering (reads)



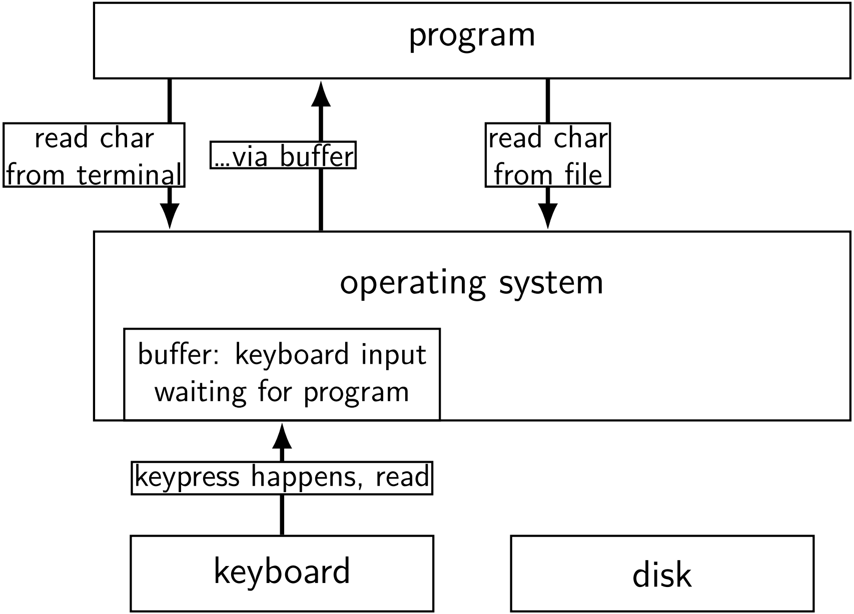
kernel buffering (reads)



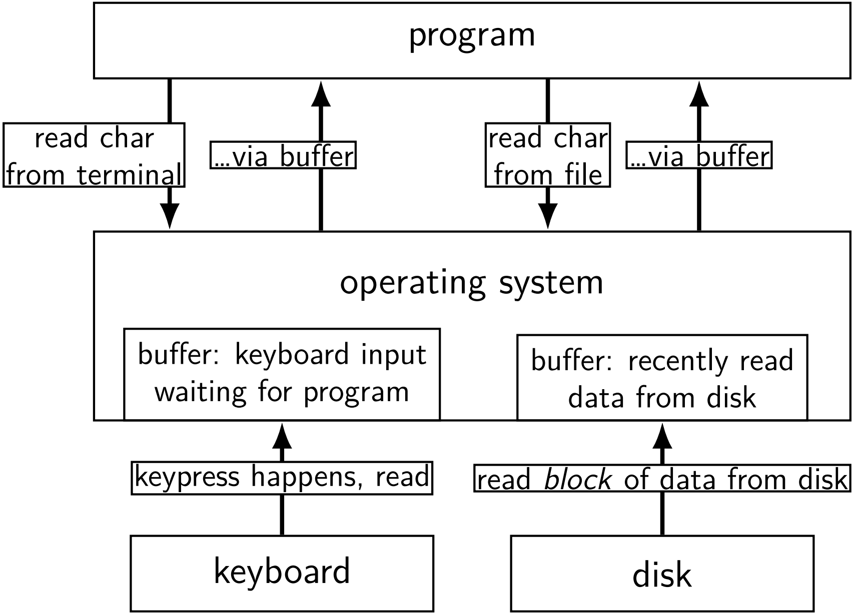
kernel buffering (reads)



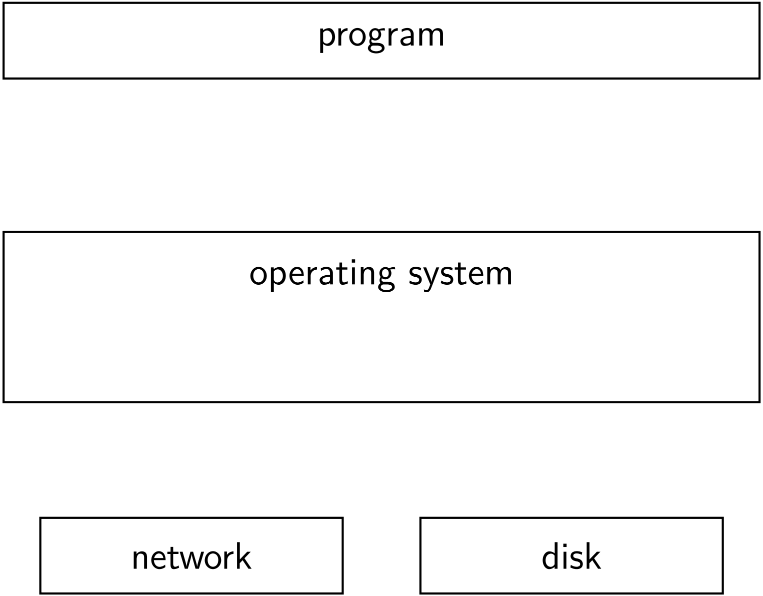
kernel buffering (reads)



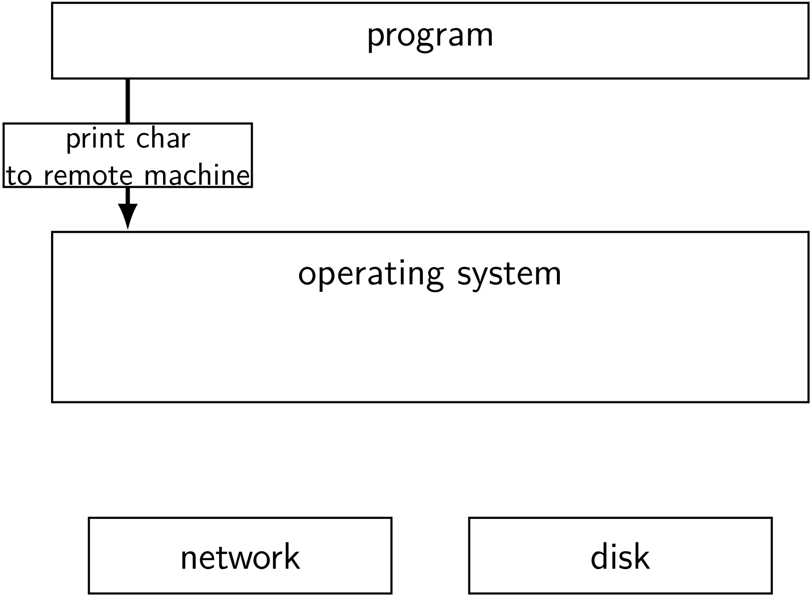
kernel buffering (reads)



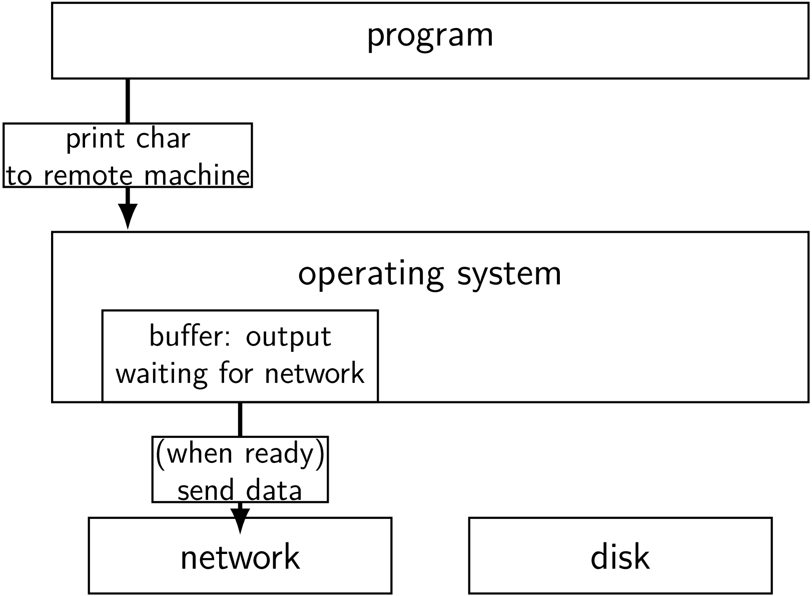
kernel buffering (writes)



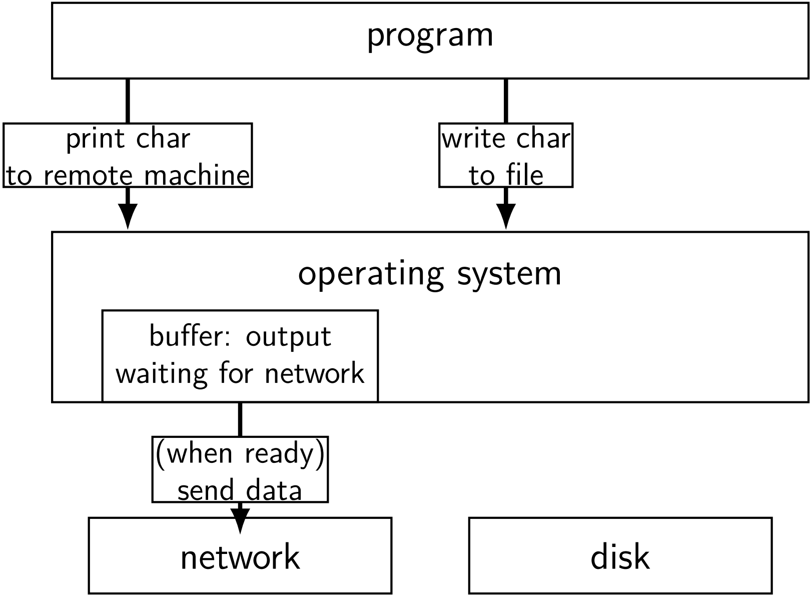
kernel buffering (writes)



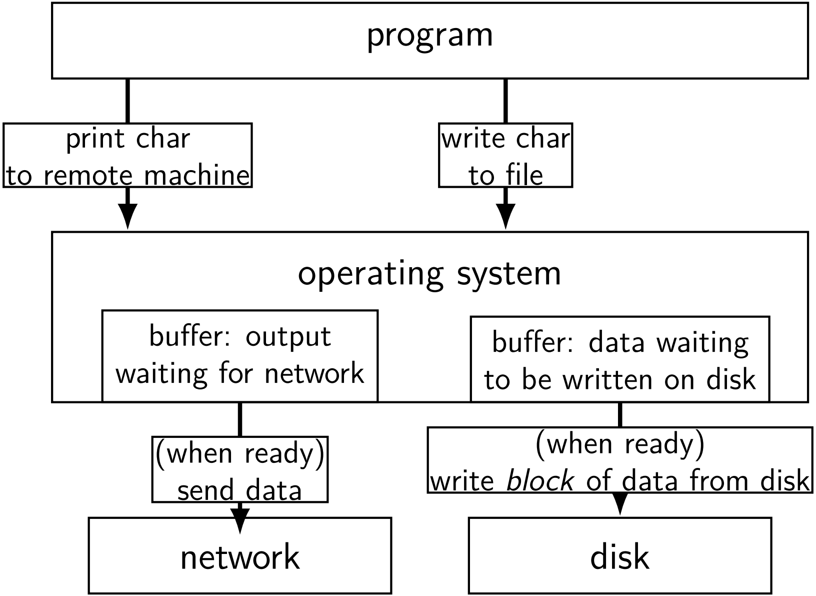
kernel buffering (writes)



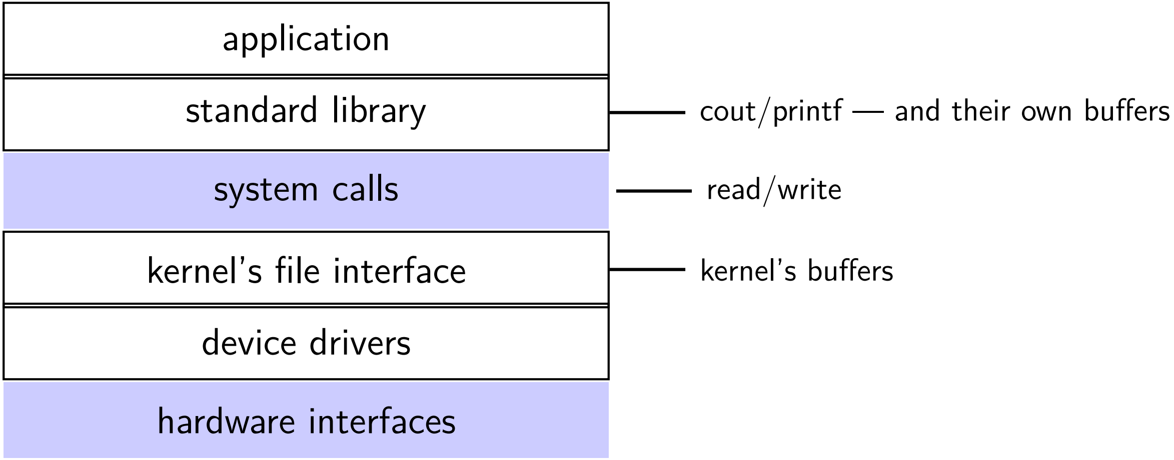
kernel buffering (writes)



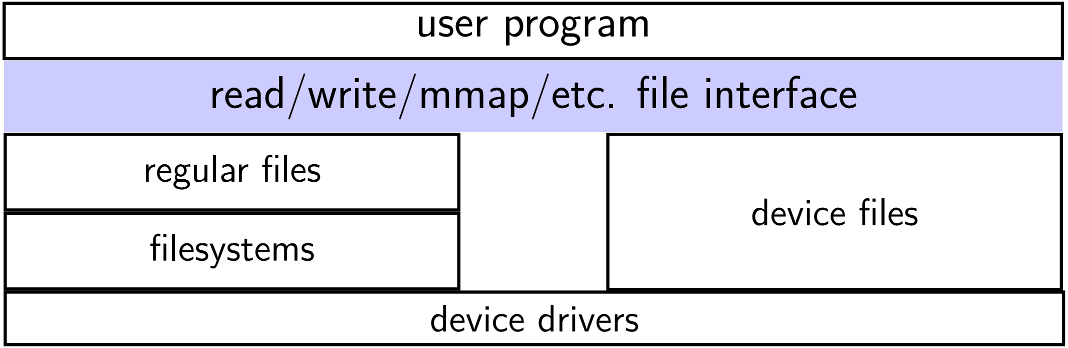
kernel buffering (writes)



layering


application
standard library
system calls
kernel's file interface
device drivers
hardware interfaces
cout/printf — and their own buffers
read/write
kernel's buffers

ways to talk to I/O devices



devices as files

talking to device? open/read/write/close

typically similar interface within the kernel

device driver implements the file interface

example device files from a Linux desktop

`/dev/snd/pcmC0D0p` — audio playback

- configure, then write audio data

`/dev/sda`, `/dev/sdb` — SATA-based SSD and hard drive

- usually access via filesystem, but can mmap/read/write directly

`/dev/input/event3`, `/dev/input/event10` — mouse and keyboard

- can read list of keypress/mouse movement/etc. events

`/dev/dri/renderD128` — builtin graphics

- DRI = direct rendering infrastructure

devices: extra operations?

read/write/mmap not enough?

- audio output device — set format of audio? headphones plugged in?
- terminal — whether to echo back what user types?
- CD/DVD — open the disk tray? is a disk present?
- ...

extra POSIX file descriptor operations:

- ioctl (general I/O control) — device driver-specific interface
- tcsetattr (for terminal settings)
- fcntl
- ...

also possibly extra device files for same device:

- `/dev/snd/controlC0` to configure audio settings for `/dev/snd/pcmC0D0p`, `/dev/snd/pcmC0D10p`, ...

Linux example: file operations

(selected subset — table of pointers to functions)

```
struct file_operations {
    ...
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
    ssize_t (*write) (struct file *, const char __user *,x
                      size_t, loff_t *);
    ...
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned lo
    ...
    int (*mmap) (struct file *, struct vm_area_struct *);
    unsigned long mmap_supported_flags;
    int (*open) (struct inode *, struct file *);
    ...
    int (*release) (struct inode *, struct file *);
    ...
};
```

special case: block devices

devices like disks often have a different interface

unlike normal file interface, works in terms of 'blocks'

- block size usually equal to page size

for working with page cache

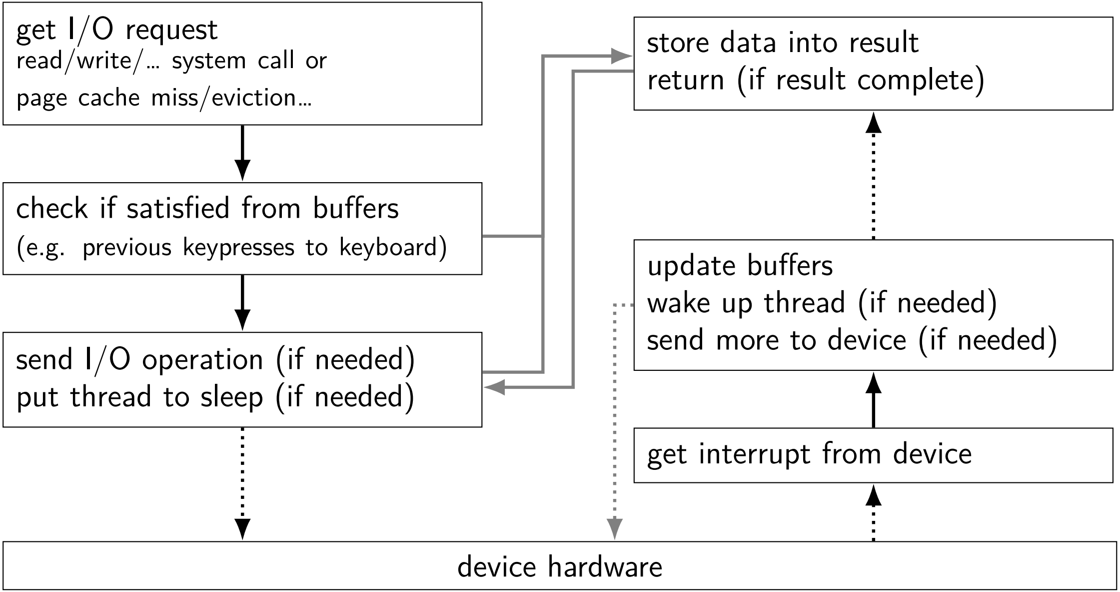
- read/write page at a time

Linux example: block device operations

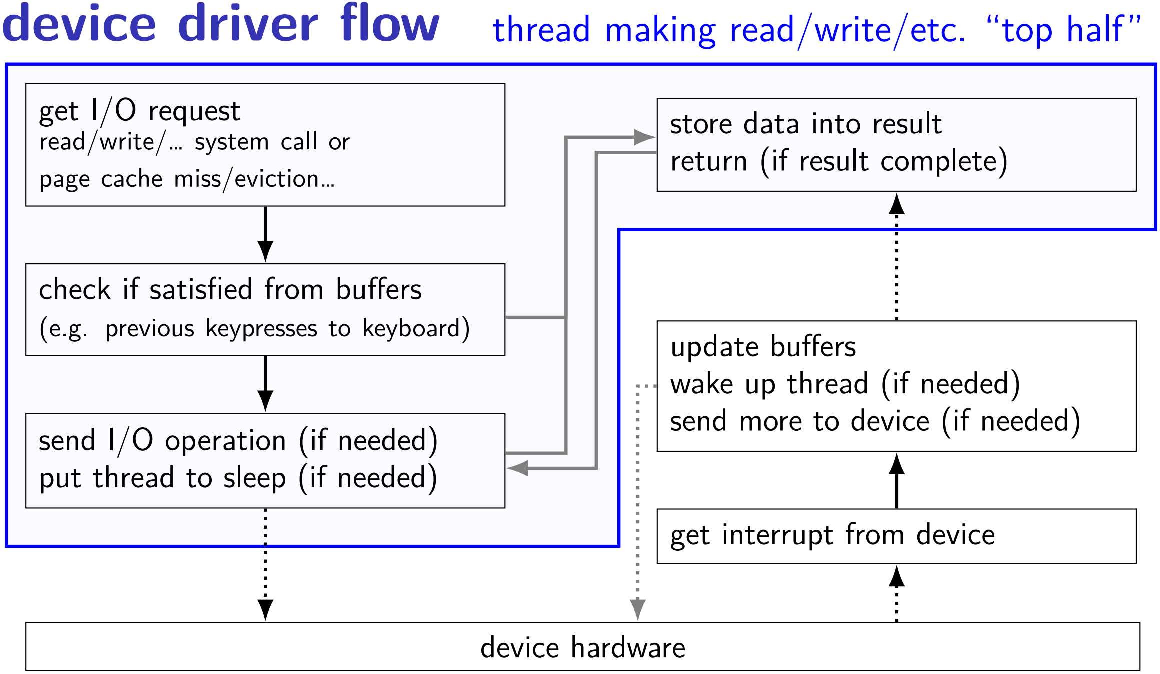
```
struct block_device_operations {
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *,
            sector_t, struct page *, bool);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, un
    ...
};
```

read/write a page for a sector number (= block number)

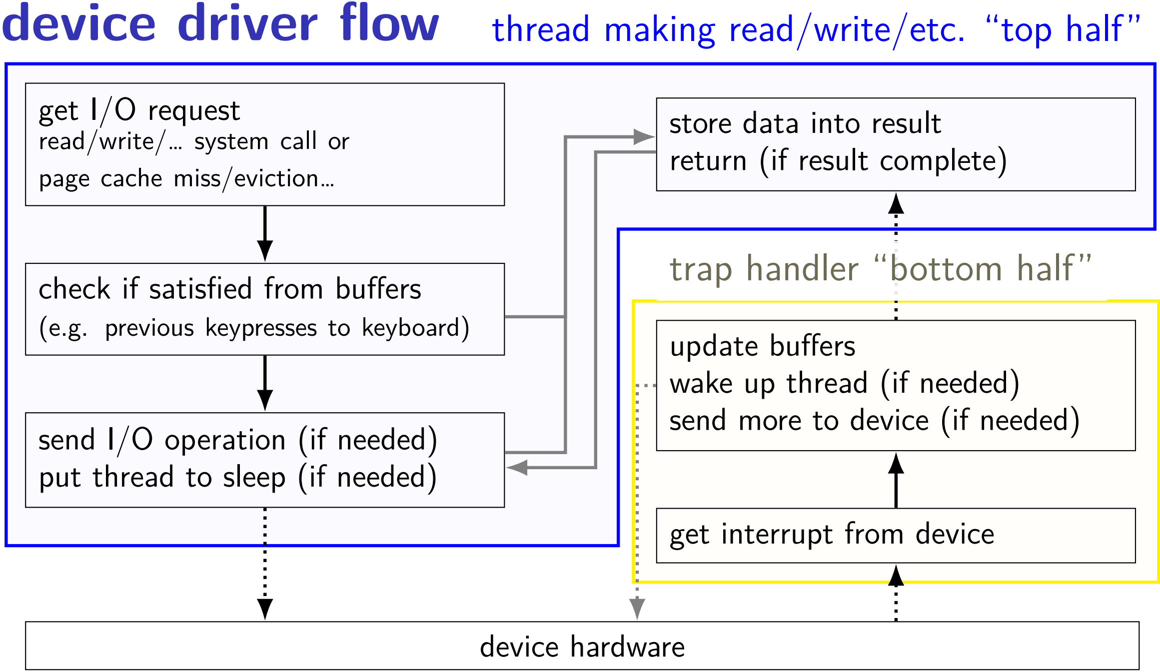
device driver flow



device driver flow



device driver flow



xv6: device files (1)

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];
```

inode = represents file on disk

pointed to by struct file referenced by fd

xv6: device files (2)

```
struct devsw {
  int (*read)(struct inode*, char*, int);
  int (*write)(struct inode*, char*, int);
};

extern struct devsw devsw[];
```

array of types of devices

special type of file on disk has index into array

- "device number"
- created via `mknod()` system call

similar scheme used on real Unix/Linux

- two numbers: major + minor device number

xv6: console devsw

code run at boot:

```
devsw[CONSOLE].write = consolewrite;
devsw[CONSOLE].read = consoleread;
```

CONSOLE is the constant 1

xv6: console devsw

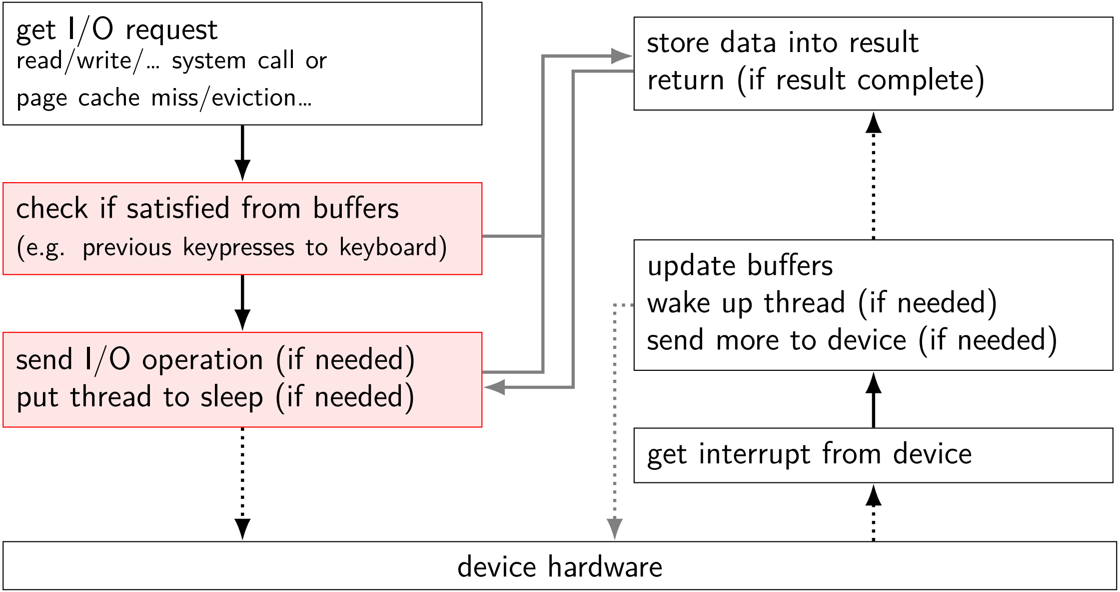
code run at boot:

```
devsw[CONSOLE].write = consolewrite;
devsw[CONSOLE].read = consoleread;
```

CONSOLE is the constant 1

consoleread/consolewrite: run when you read/write console

device driver flow



xv6: console top half (read)

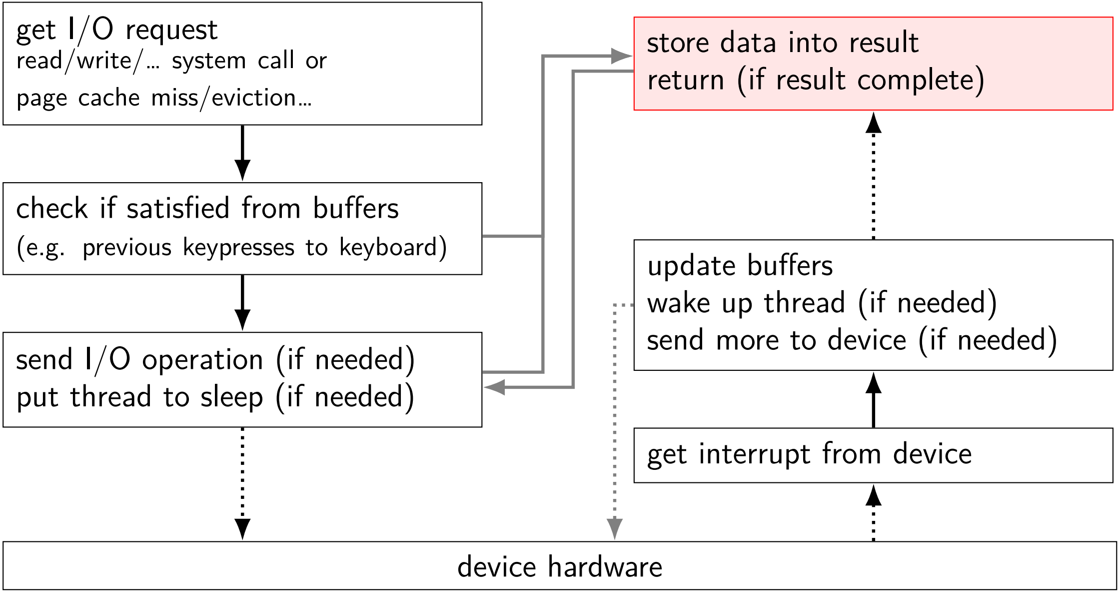
```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    while(input.r == input.w){
      if(myproc()->killed){
        ...
        return -1;
      }
      sleep(&input.r, &cons.lock);
    }
    ...
  }
  release(&cons.lock)
  ...
}
```

if at end of buffer

r = reading location, w = writing location

put thread to sleep

device driver flow



xv6: console top half (read)

```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    ...
    c = input.buf[input.r++ % INPUT_B
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

copy from kernel buffer
to user buffer (passed to read)

xv6: console top half (read)

```
int
consoleread(struct inode *ip, char *dst, int n)
{
  ...
  target = n;
  acquire(&cons.lock);
  while(n > 0){
    ...
    c = input.buf[input.r++ % INPUT_B
    ...
    *dst++ = c;
    --n;
    if (c == '\n')
      break;
  }
  release(&cons.lock)
  ...
  return target - n;
}
```

copy from kernel buffer
to user buffer (passed to read)

xv6: console top half

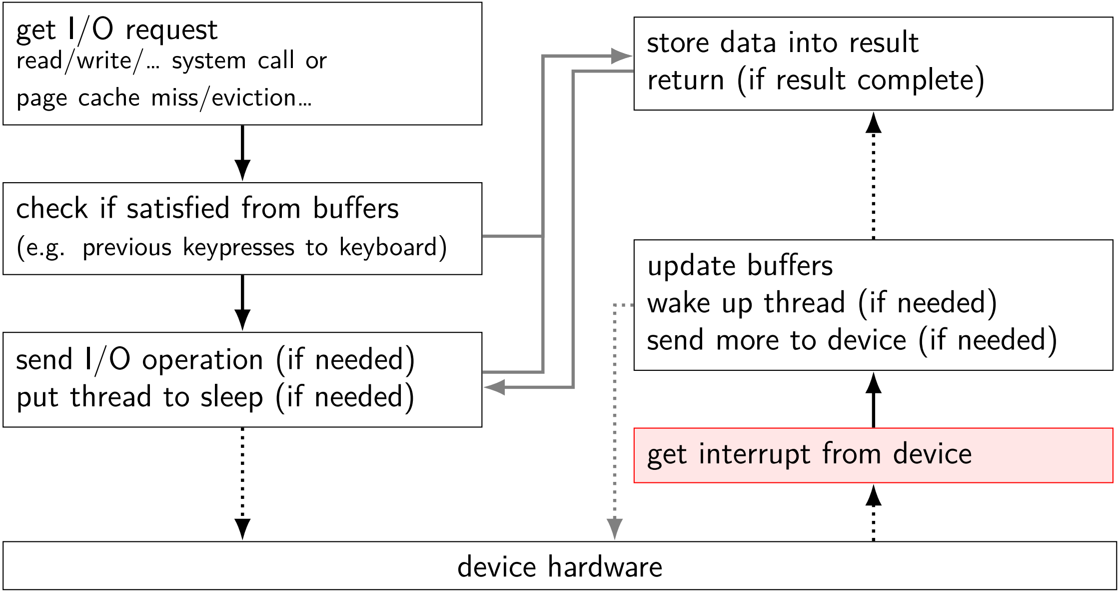
wait for buffer to fill

- no special work to request data — keyboard input always sent

copy from buffer

check if done (newline or enough chars), if not repeat

device driver flow



xv6: console interrupt (one case)

```
void
trap(struct trapframe *tf) {
  ...
  switch(tf->trapno) {
    ...
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapcieoi();
    break;
    ...
  }
  ...
}
```

kbdintr: actually read from keyboard device

lapcieoi: tell CPU “I’m done with this interrupt”

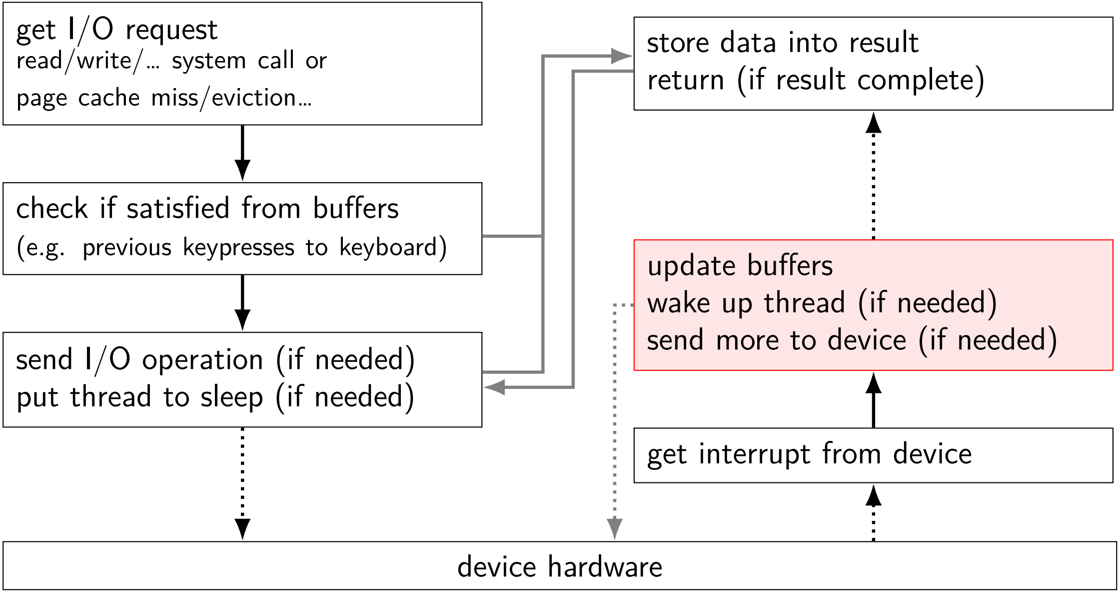
xv6: console interrupt (one case)

```
void
trap(struct trapframe *tf) {
  ...
  switch(tf->trapno) {
    ...
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapcieoi();
    break;
    ...
  }
  ...
}
```

kbdintr: actually read from keyboard device

lapcieoi: tell CPU “I’m done with this interrupt”

device driver flow



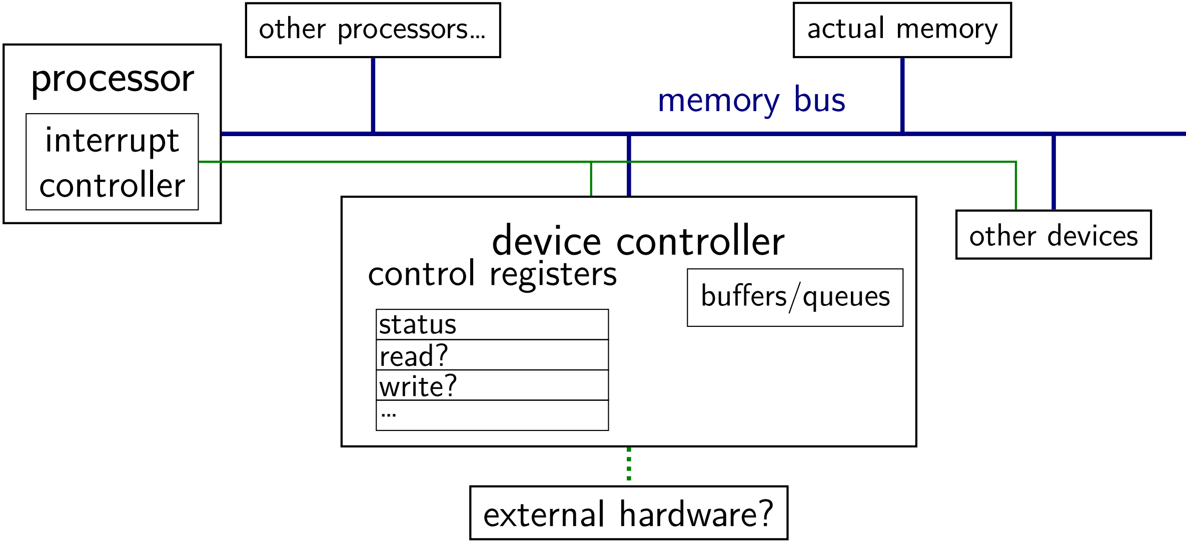
xv6: console interrupt reading

kbdintr fuction actually reads from device

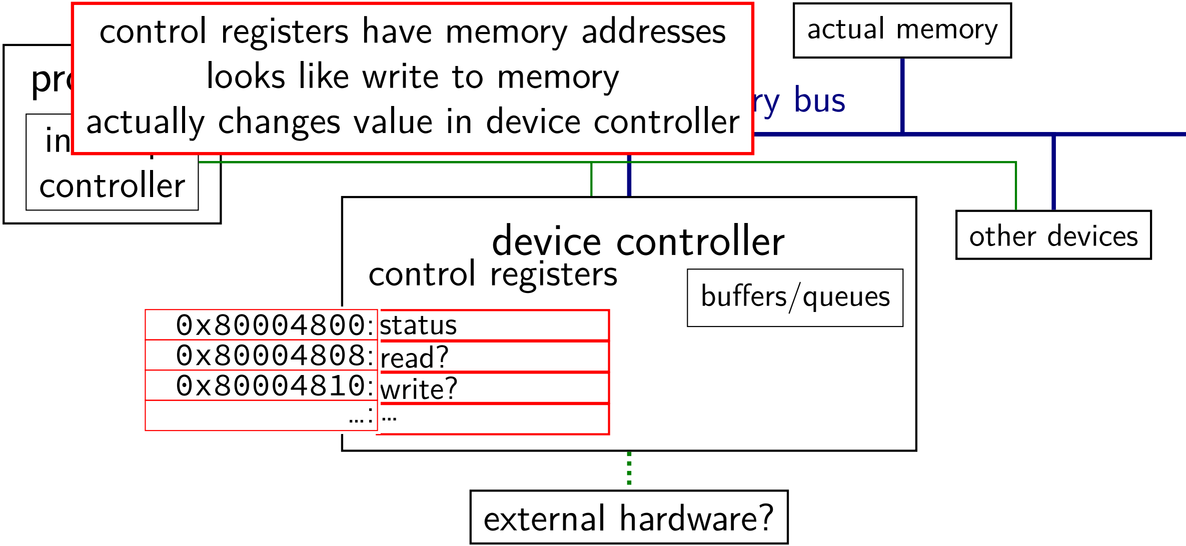
adds data to buffer (if room)

wakes up sleeping thread (if any)

connecting devices

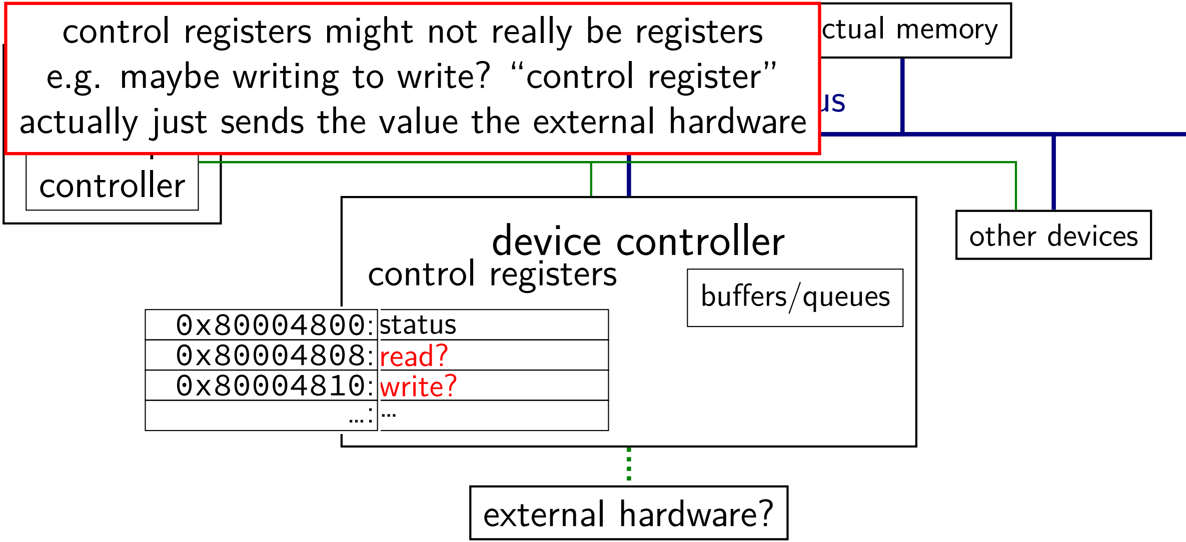


connecting devices

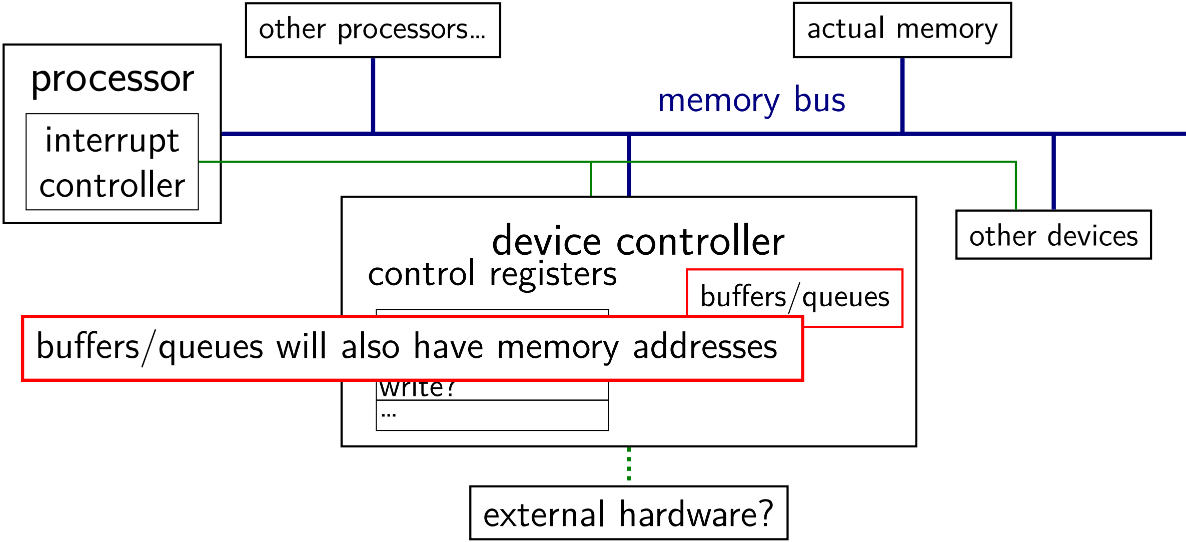


connecting devices

control registers might not really be registers
e.g. maybe writing to write? "control register"
actually just sends the value the external hardware

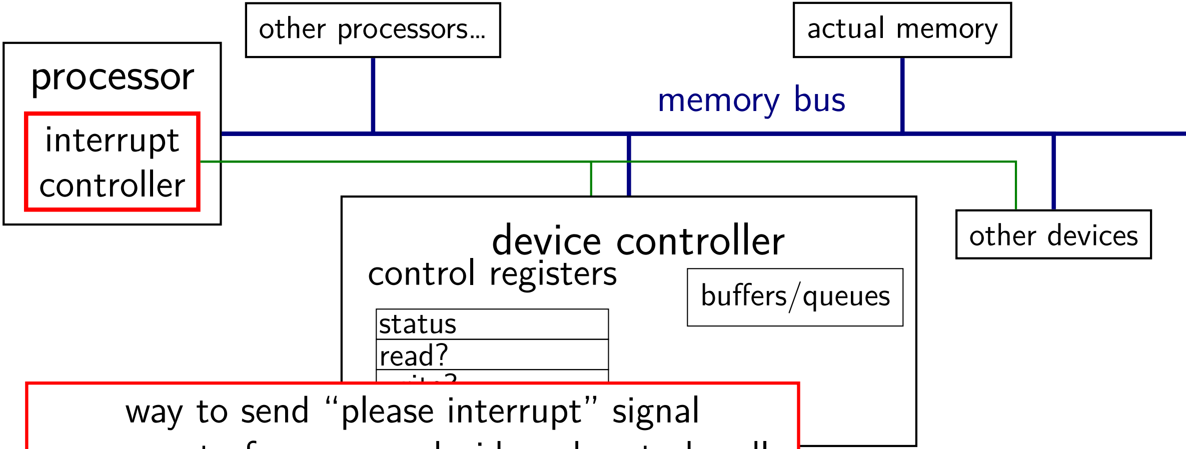


connecting devices



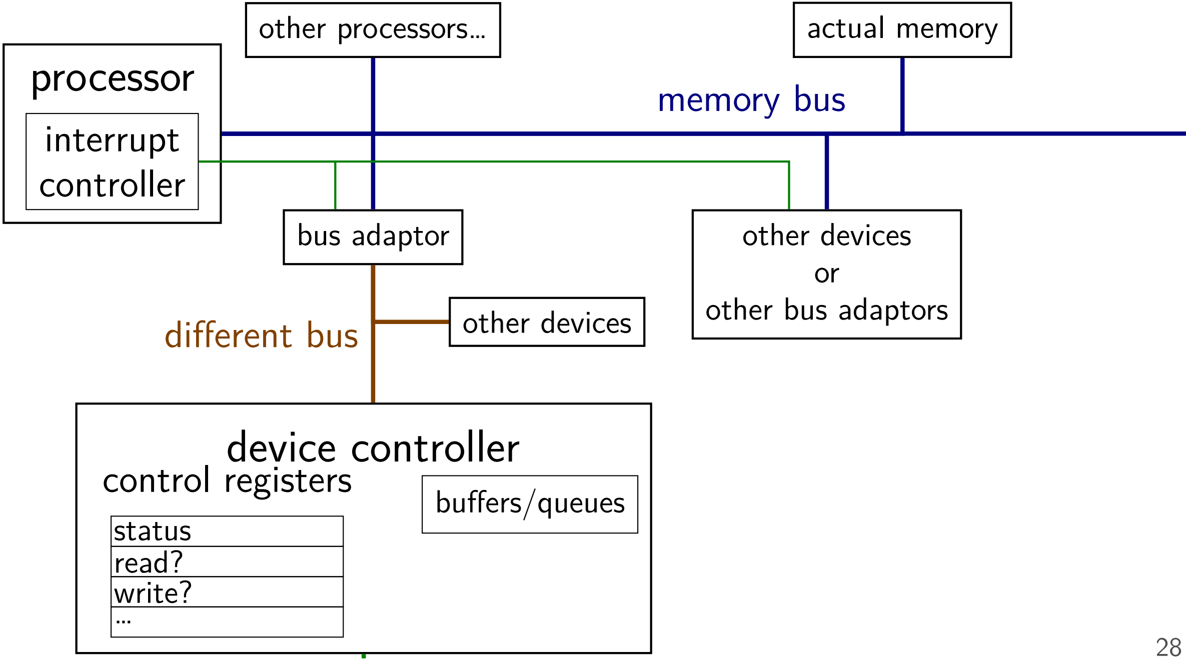
buffers/queues will also have memory addresses

connecting devices



way to send "please interrupt" signal
component of processor decides when to handle
(deals with ordering, interrupt disabling,
which of several processors handles it, ..., etc.)

bus adaptors



devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

devices as magic memory (1)

devices expose memory locations to read/write

use read/write instructions to manipulate device

example: keyboard controller

read from magic memory location — get last keypress/release

reading location clears buffer for next keypress/release

get interrupt whenever new keypress/release you haven't read

device as magic memory (2)

example: display controller

write to pixels to magic memory location — displayed on screen

other memory locations control format/screen size

example: network interface

write to buffers

write "send now" signal to magic memory location — send data

read from "status" location, buffers to receive

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

...I press ‘e’, OS reads what?

what about caching?

caching “last keypress/release”?

I press ‘h’, OS reads ‘h’, does that get cached?

...I press ‘e’, OS reads what?

solution: OS can mark memory uncachable

x86: bit in page table entry can say “no caching”

aside: I/O space

x86 has a "I/O addresses"

like memory addresses, but accessed with different instruction
- `in` and `out` instructions

historically — and sometimes still: separate I/O bus

more recent processors/devices usually use memory addresses
- no need for more instructions, buses
- always have layers of bus adaptors to handle compatibility issues
- other reasons to have devices and memory close (later)

xv6 keyboard access

two control registers:

KBSTATP: status register (I/O address `0x64`)
KBDATAP: data buffer (I/O address `0x60`)

```
// inb() runs 'in' instruction: read from I/O address
st = inb(KBSTATP);
// KBS_DIB: bit indicates data in buffer
if ((st & KBS_DIB) == 0)
  return −1;
data = inb(KBDATAP);  // read from data --- *clears* buffer

/* interpret data to learn what kind of keypress/release */
```

programmed I/O

"programmed I/O": write to or read from device controller buffers directly

OS runs loop to transfer data to or from device controller

might still be triggered by interrupt

- new data in buffer to read?
- device processed data previously written to buffer?

exercise

system is running two applications

A: reading from network

B: doing tons of computation

timeline:

A calls read() to 8KB of data from network

16KB of data comes in 10ms later

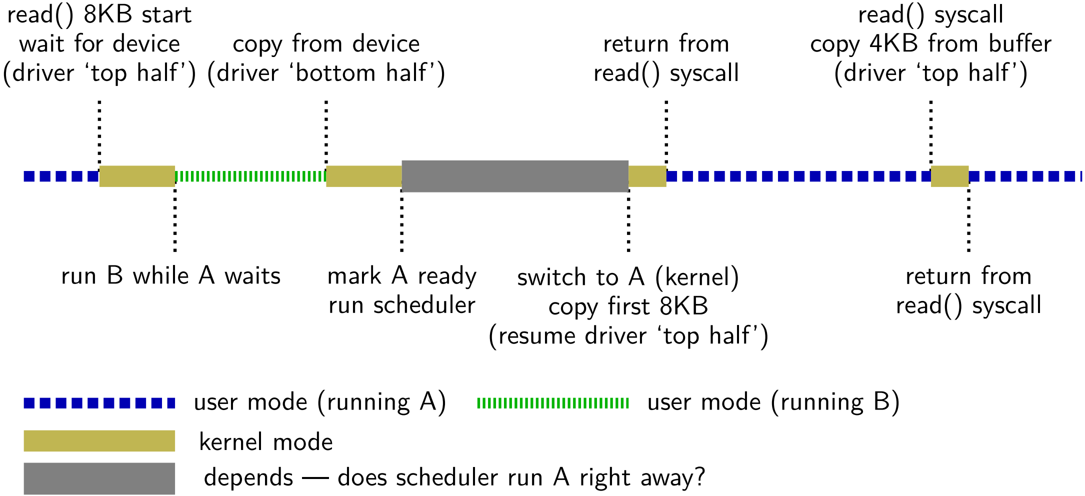
A calls read() again to get 4KB more

exercise 1: how many kernel/user mode switches?

exercise 2: how many context switches?

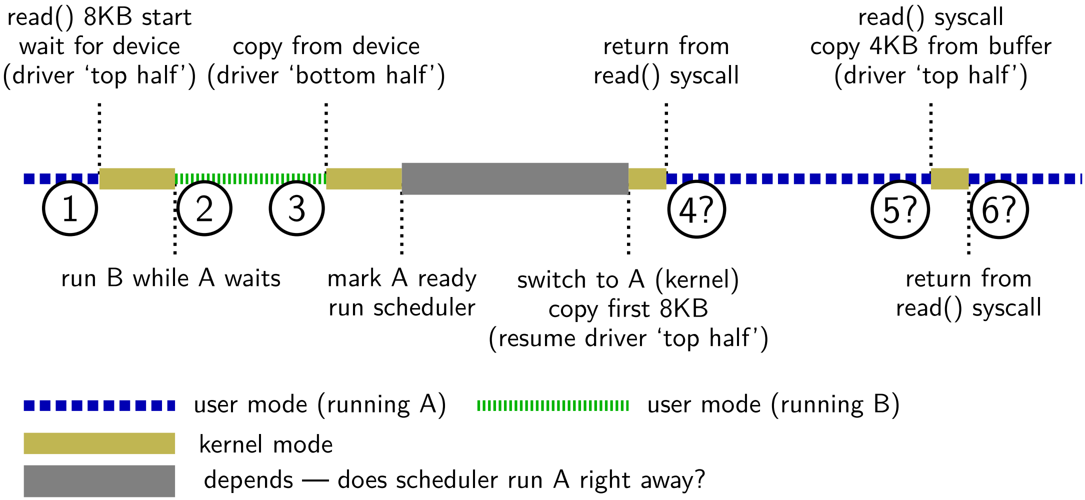
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get 4KB more



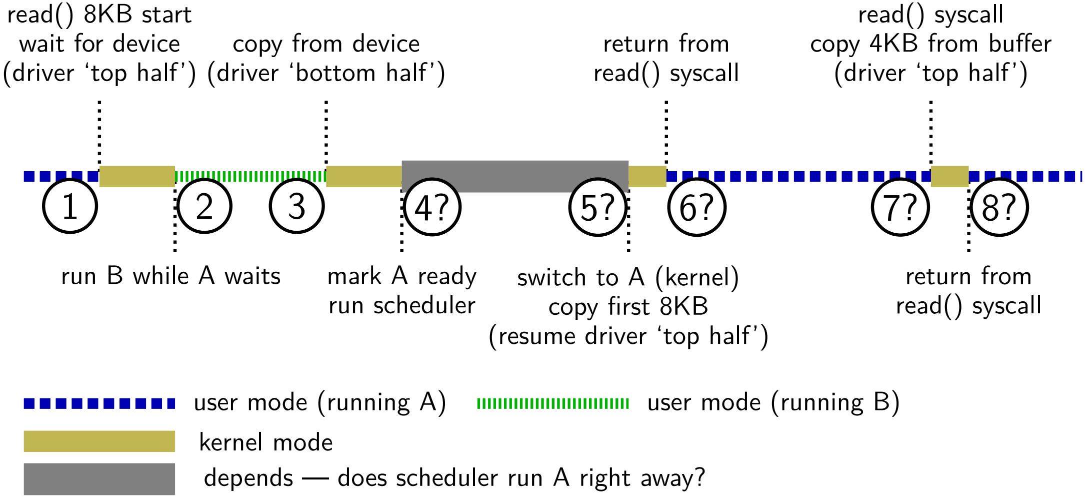
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get 4KB more



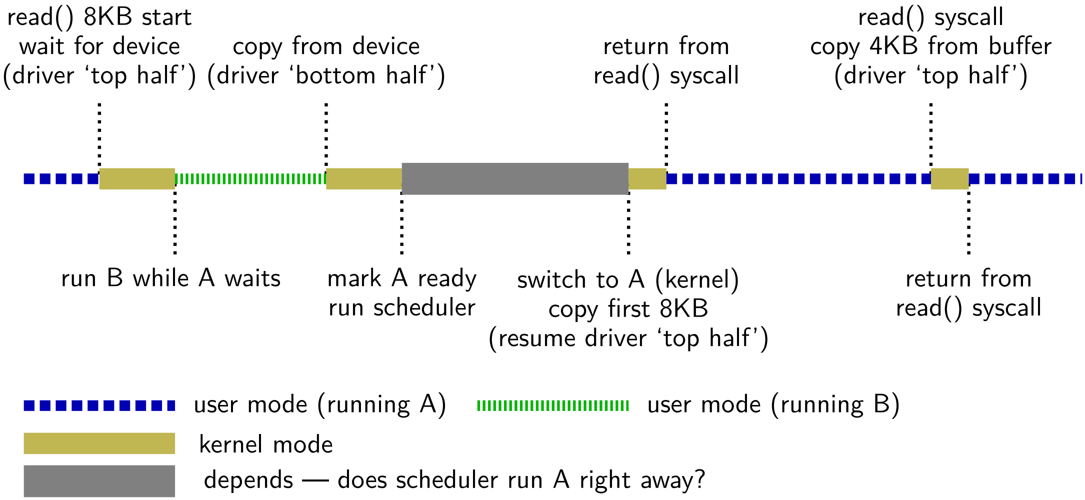
how many mode switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get 4KB more



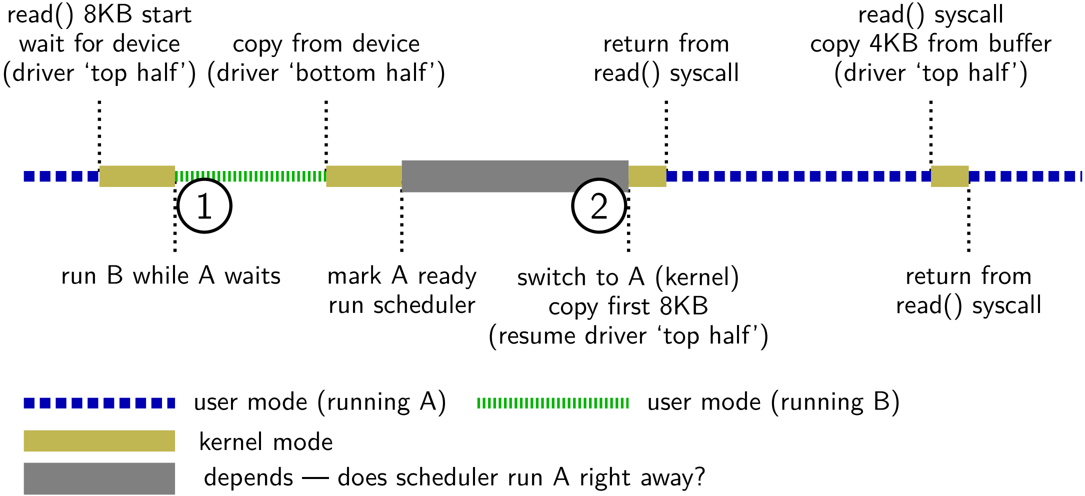
how many context switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB

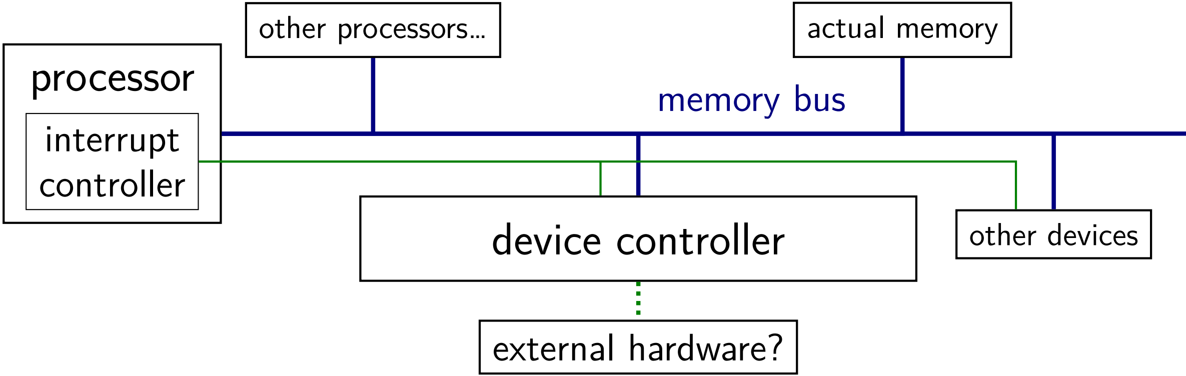


how many context switches?

A calls read() to 8KB of data from network
16KB of data comes in 10ms later
A calls read() again to get remaining 4KB



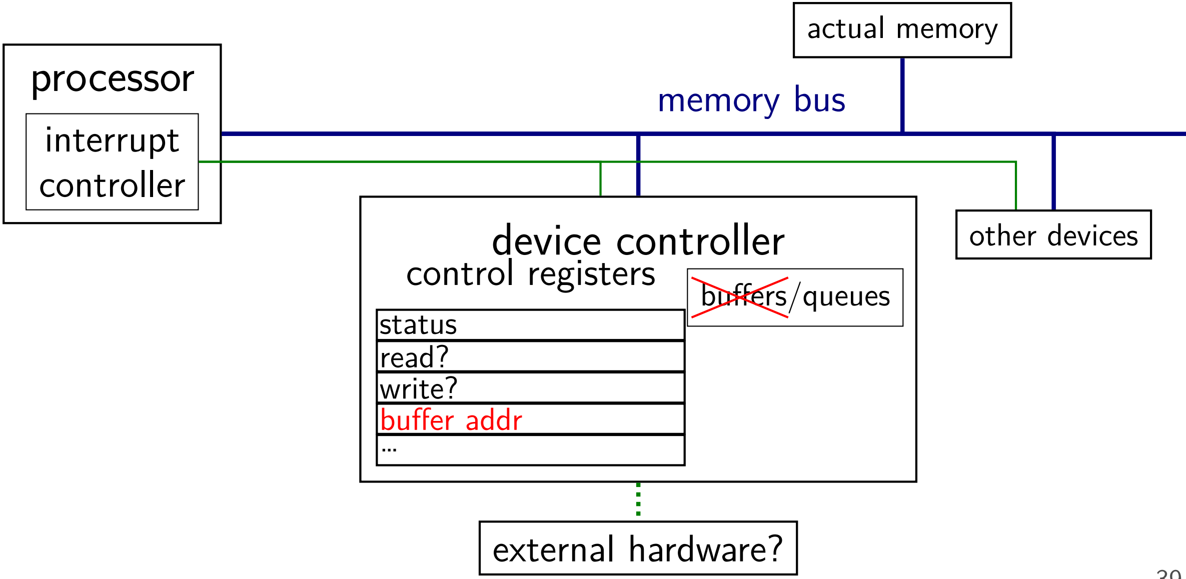
direct memory access (DMA)



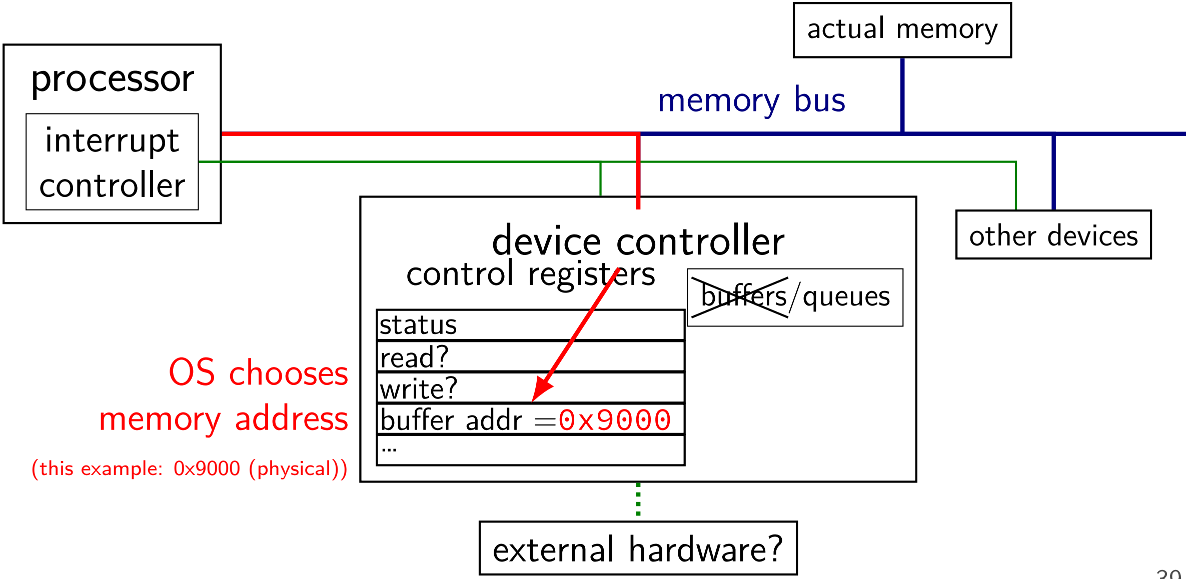
observation: devices can read/write memory

can have device copy data to/from memory

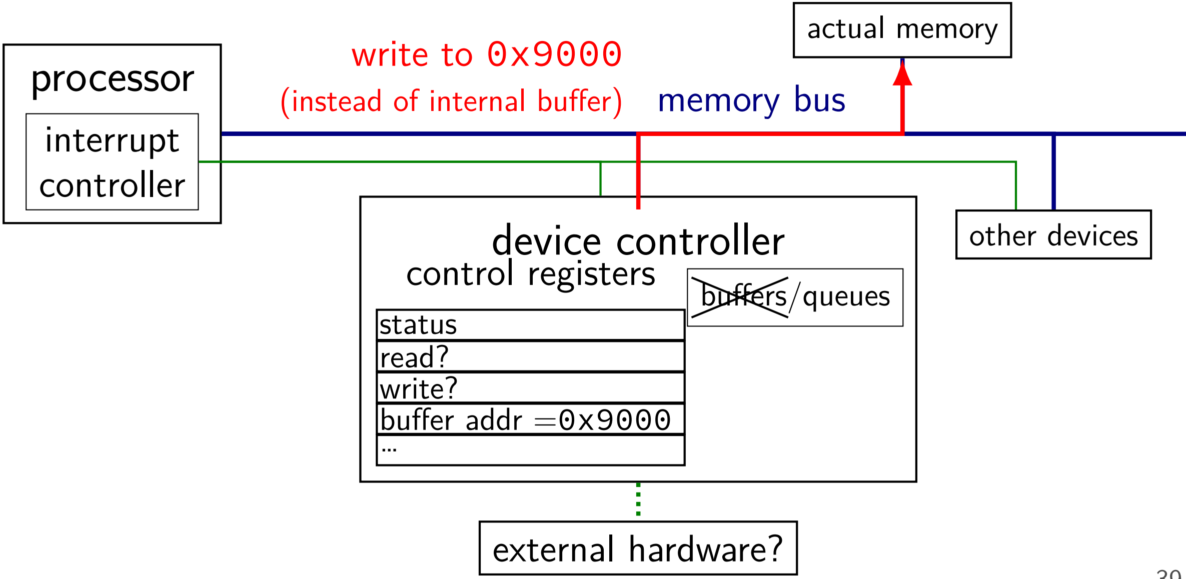
direct memory access (DMA)



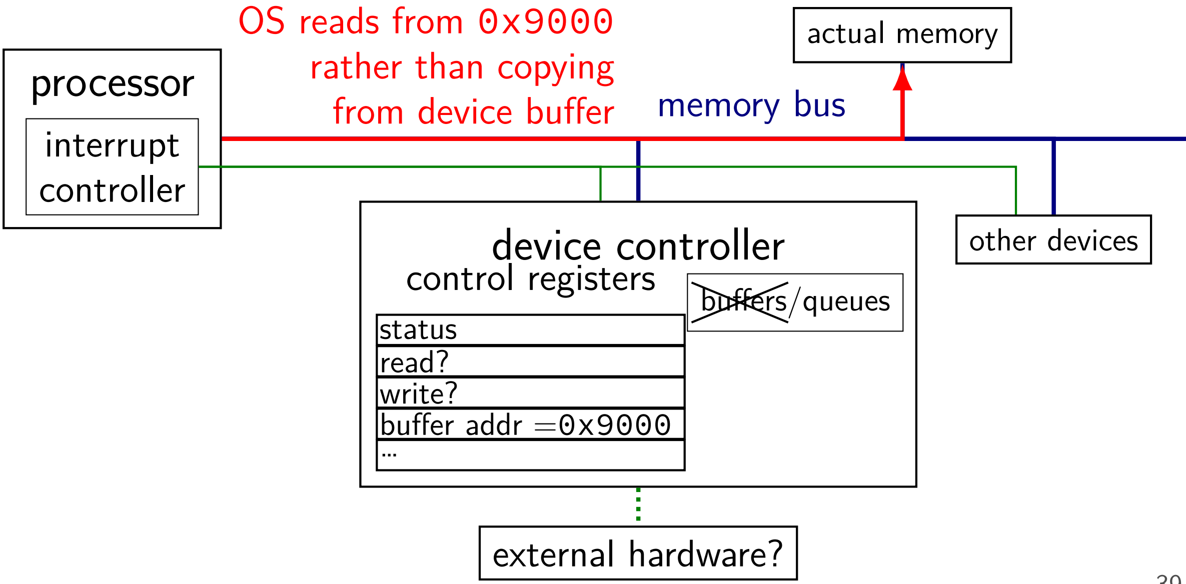
direct memory access (DMA)



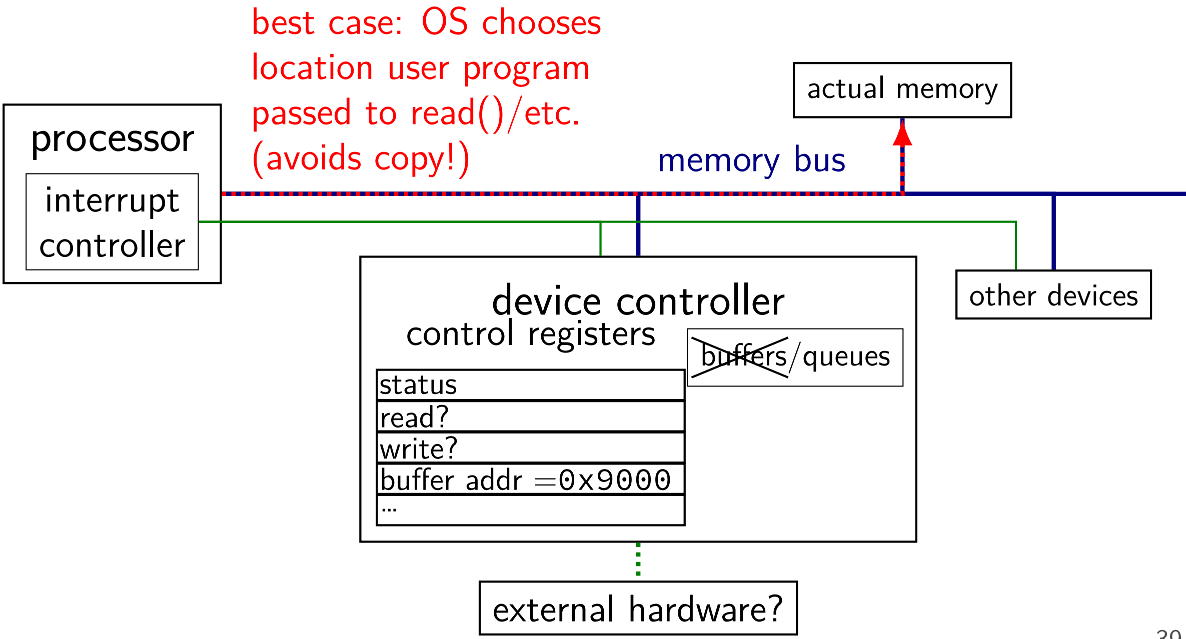
direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)



direct memory access (DMA)

much faster, e.g., for disk or network I/O

avoids having processor run a loop to copy data
- OS can run normal program during data transfer
- interrupt tells OS when copy finished

device uses memory as very large buffer space

device puts data where OS wants it directly (maybe)
- OS specifies physical address to use…
- instead of reading from device controller

direct memory access (DMA)

much faster, e.g., for disk or network I/O

avoids having processor run a loop to copy data
- OS can run normal program during data transfer
- interrupt tells OS when copy finished

device uses memory as very large buffer space

device puts data where OS wants it directly (maybe)
- OS specifies physical address to use…
- instead of reading from device controller

OS puts data where it wants

so far: where it wants is the device driver's buffer

OS puts data where it wants

so far: where it wants is the device driver's buffer

seems like OS could also put it directly where application wants it?

- i.e. pointer passed to read() system call
- called "zero-copy I/O"

OS puts data where it wants

so far: where it wants is the device driver's buffer

seems like OS could also put it directly where application wants it?

- i.e. pointer passed to read() system call
- called "zero-copy I/O"

should be faster, but, in practice, very rarely done:

- if part of regular file, can't easily share with page cache
- device might expect contiguous physical addresses
- device might expect physical address is at start of physical page
- device might write data in differnt format than application expects
- device might read too much data
- need to deal with application exiting/being killed before device finishes
- ...

devices summary

device *controllers* connected via memory bus

- usually assigned physical memory addresses
- sometimes separate "I/O addresses" (special load/store instructions)

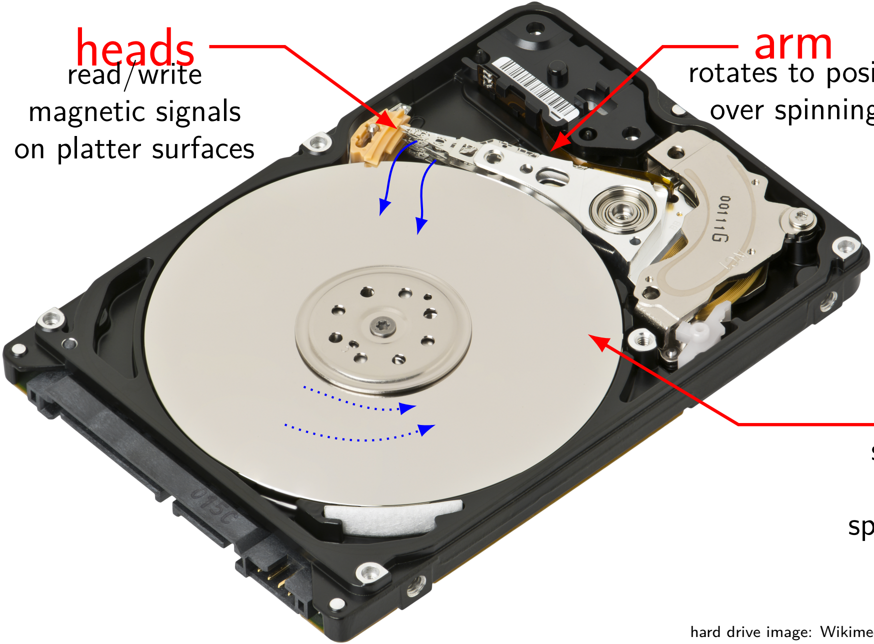
controller looks like "magic memory" to OS

- load/store from device controller registers like memory
- setting/reading control registers can trigger device operations

two options for data transfer

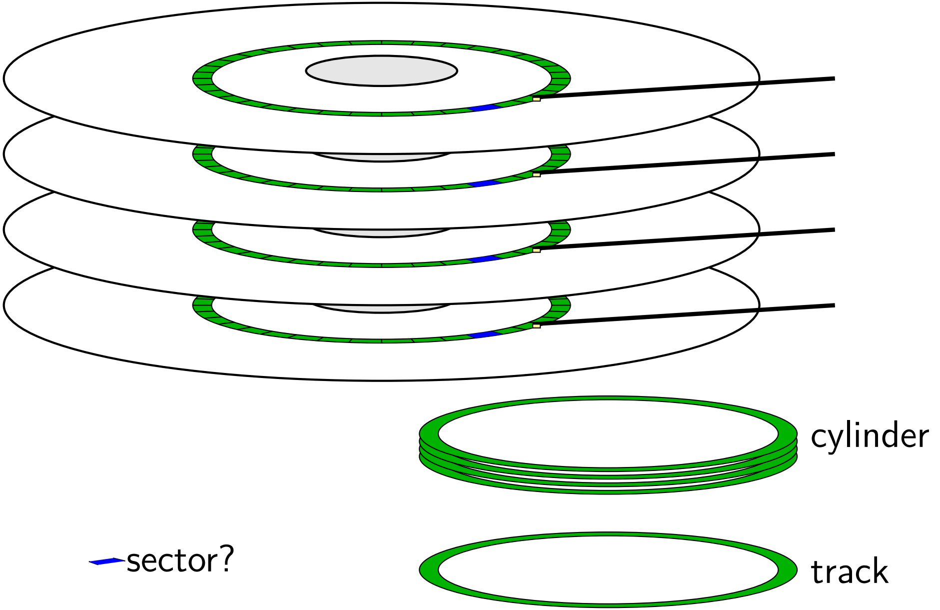
- programmed I/O: OS reads from/writes to buffer within device controller
- direct memory access (DMA): device controller reads/writes normal memory

hard drives



hard drive image: Wikimedia Commons / Evan-Amos

sectors/cylinders/etc.

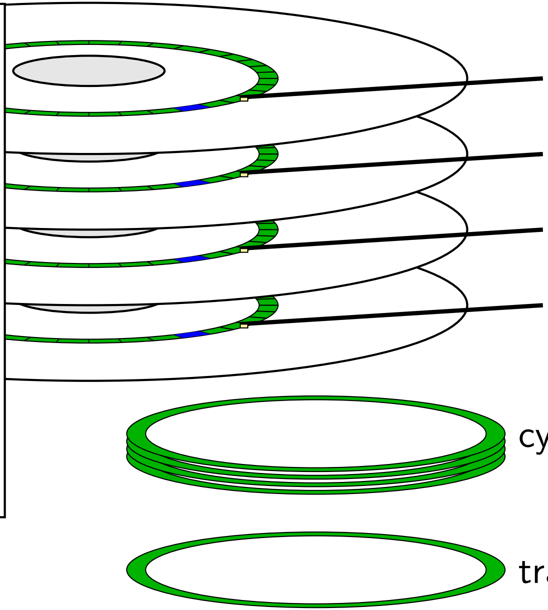


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

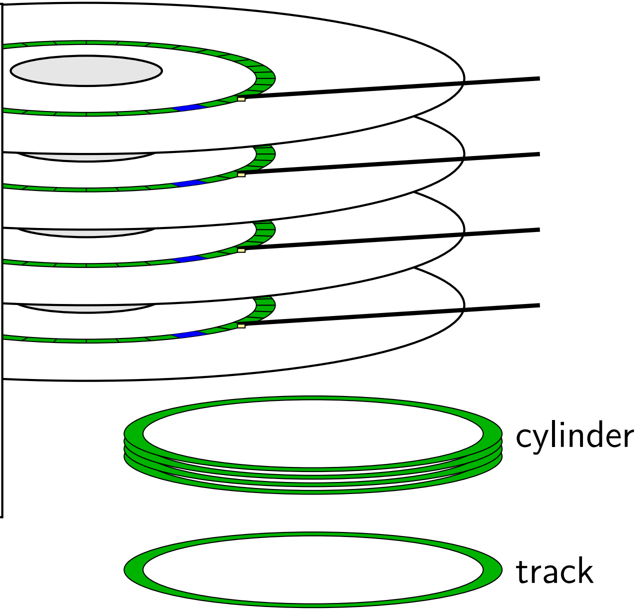
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data

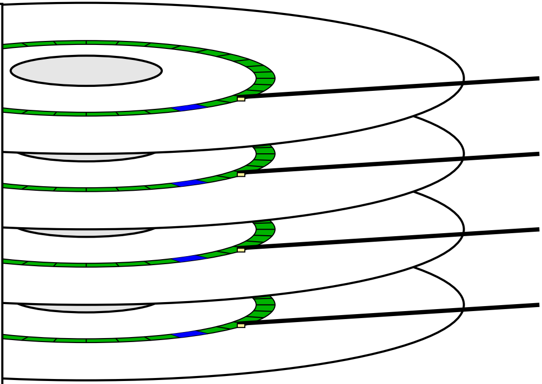


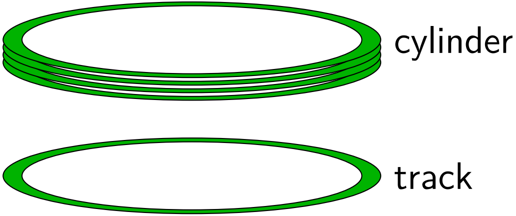
sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



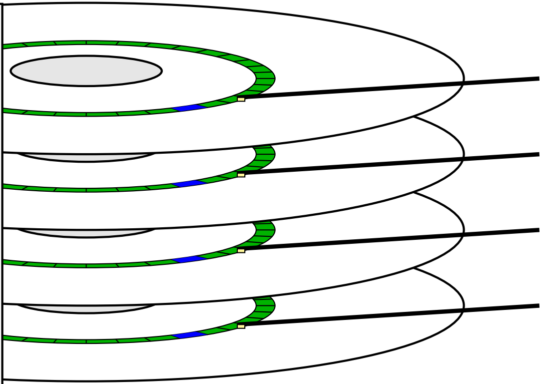


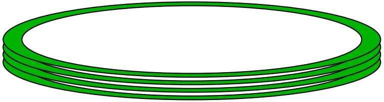
sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

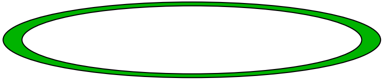
rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data





cylinder



track

sector?

OS to disk interface

disk takes read/write requests

- sector number(s)
- location of data for sector
- modern disk controllers: typically direct memory access

typically: close sector numbers $\implies$ close on disk

- for spinning disks, faster to read/write together
- for SSDs, doesn't matter much

can have queue of pending requests

disk processes them in some order

- OS can say "write X before Y"

the FAT filesystem

FAT: File Allocation Table

probably simplest widely used filesystem (family)

named for important data structure: *file allocation table*

FAT and sectors

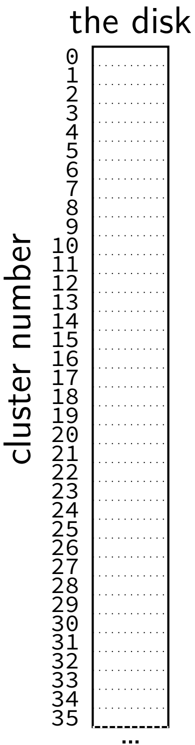
FAT divides disk into *clusters*

composed of one or more sectors

sector = minimum amount hardware can read

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



FAT and sectors

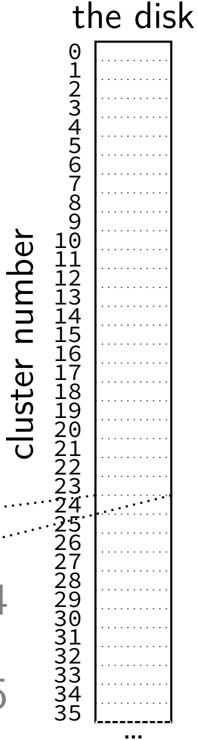
FAT divides disk into *clusters*

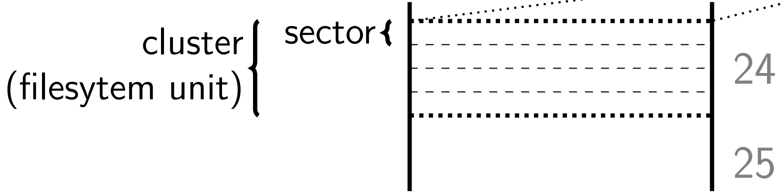
composed of one or more sectors

sector = minimum amount hardware can read

- determined by disk hardware
- historically 512 bytes, but often bigger now

cluster: typically 512 to 4096 bytes



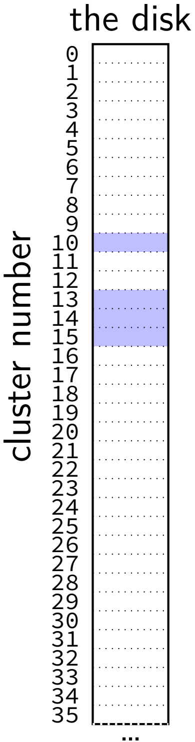


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find the list of clusters

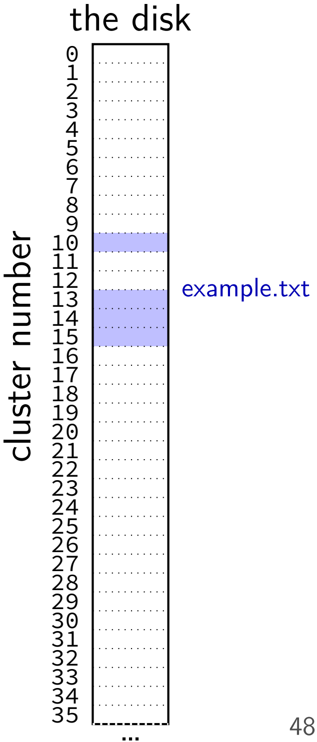


FAT: clusters and files

a file's data stored in a list of clusters

file size isn't multiple of cluster size? waste space

reading a file? need to find the list of clusters



FAT: the file allocation table

big array on disk, one entry per cluster

each entry contains a number — usually “next cluster”

cluster num.	entry value
0	4
1	7
2	5
3	1434
...	...
1000	4503
1001	1523
...	...

backup slides

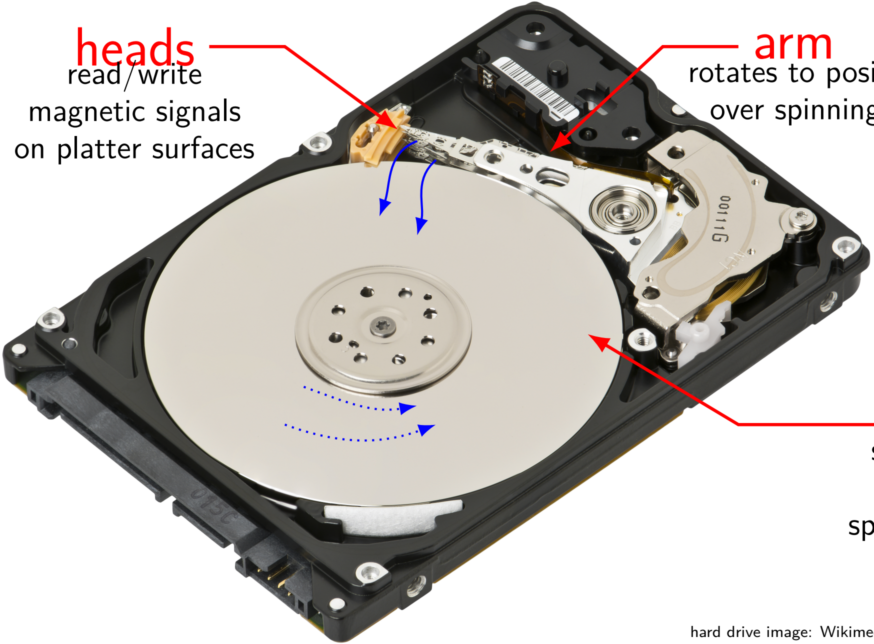
why hard drives?

what filesystems were designed for

currently most cost-effective way to have a lot of online storage

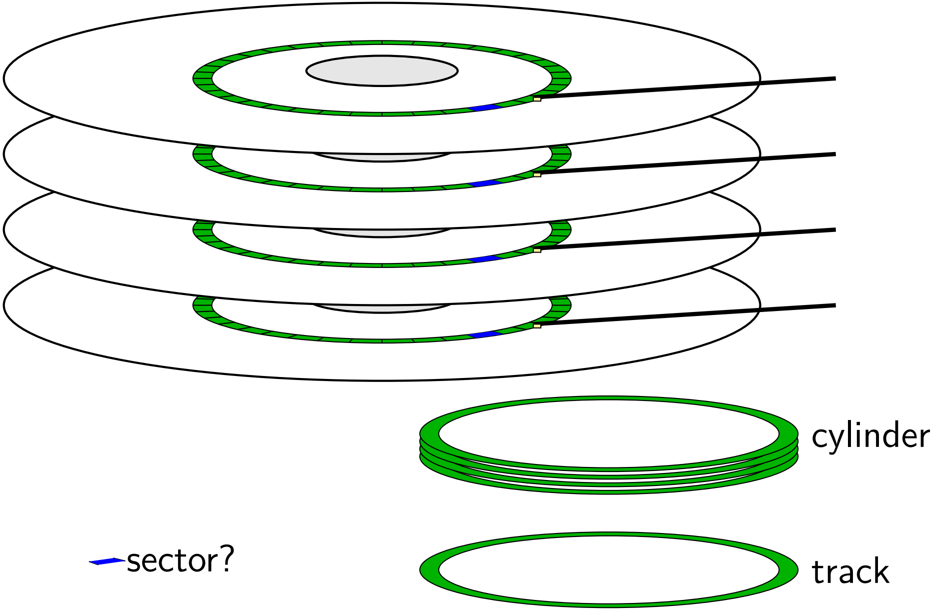
solid state drives (SSDs) imitate hard drive interfaces

hard drives



hard drive image: Wikimedia Commons / Evan-Amos

sectors/cylinders/etc.

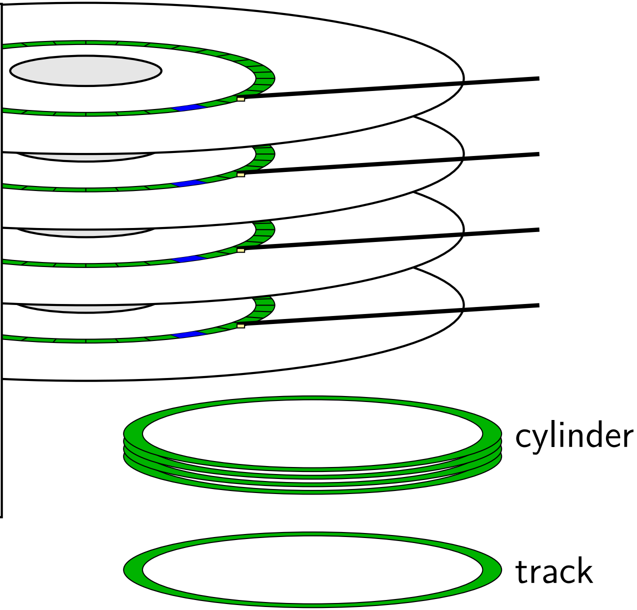


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



sector?

sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

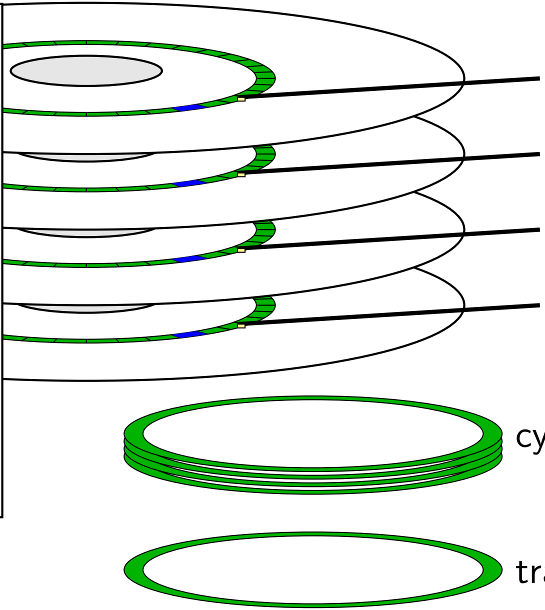
rotate platter to sector

depends on rotation speed

faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



sectors/cylinders/etc.

seek time — 5–10ms

move heads to cylinder

faster for adjacent accesses

rotational latency — 2–8ms

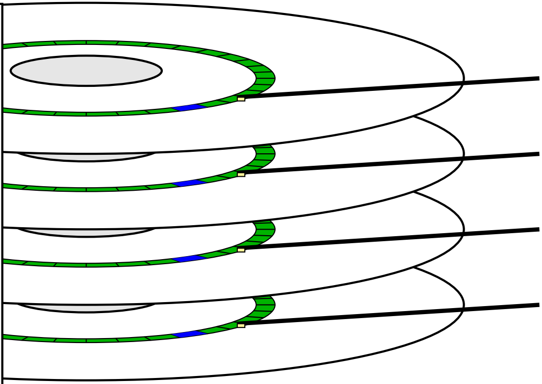
rotate platter to sector

depends on rotation speed

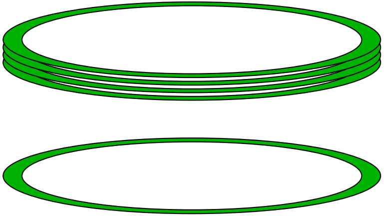
faster for adjacent reads

transfer time — 50–100+MB/s

actually read/write data



sector?

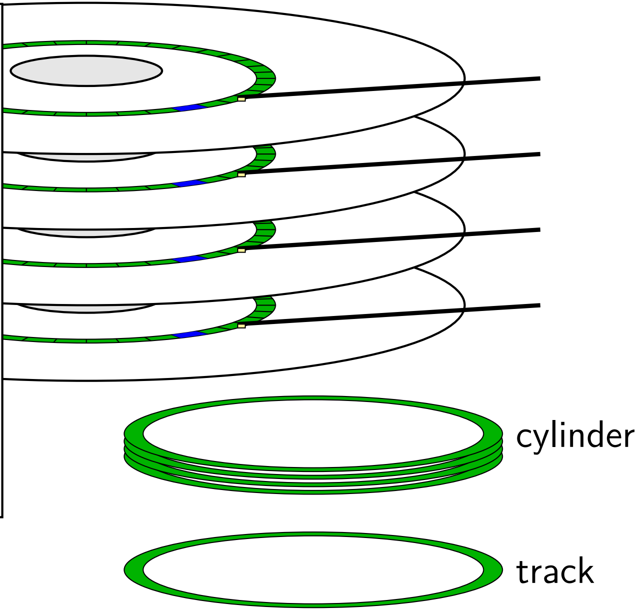


sectors/cylinders/etc.

seek time — 5–10ms
move heads to cylinder
faster for adjacent accesses

rotational latency — 2–8ms
rotate platter to sector
depends on rotation speed
faster for adjacent reads

transfer time — 50–100+MB/s
actually read/write data



sector?

disk latency components

queue time — how long read waits in line?

- depends on number of reads at a time, scheduling strategy

disk controller/etc. processing time

seek time — head to cylinder

rotational latency — platter rotate to sector

transfer time

cylinders and latency

cylinders closer to edge of disk are faster (maybe)

less rotational latency

sector numbers

historically: OS knew cylinder/head/track location

now: opaque sector numbers

- more flexible for hard drive makers
- same interface for SSDs, etc.

typical pattern: low sector numbers = probably closer to edge (faster?)

typical pattern: adjacent sector numbers = adjacent on disk

actual mapping: decided by disk controller

hard disks are unreliable

Google study (2007), heavily utilized cheap disks

1.7% to 8.6% annualized failure rate

- varies with age
- $\approx$ chance a disk fails each year
- disk fails $=$ needs to be replaced

9% of working disks had reallocated sectors

bad sectors

modern disk controllers do sector remapping

part of physical disk becomes bad — use a different one

- disk uses error detecting code to tell data is bad
- similar idea to storing + checking hash of data

this is expected behavior

maintain mapping (special part of disk, probably)

queuing requests

recall: multiple active requests

queue of reads/writes

- in disk controller *and/or* OS

disk is faster for adjacent/close-by reads/writes

- less seek time/rotational latency

disk controller and/or OS may need *schedule* requests

- group nearby requests together

as user of disk: better to request multiple things at a time

disk performance and filesystems

filesystem can...

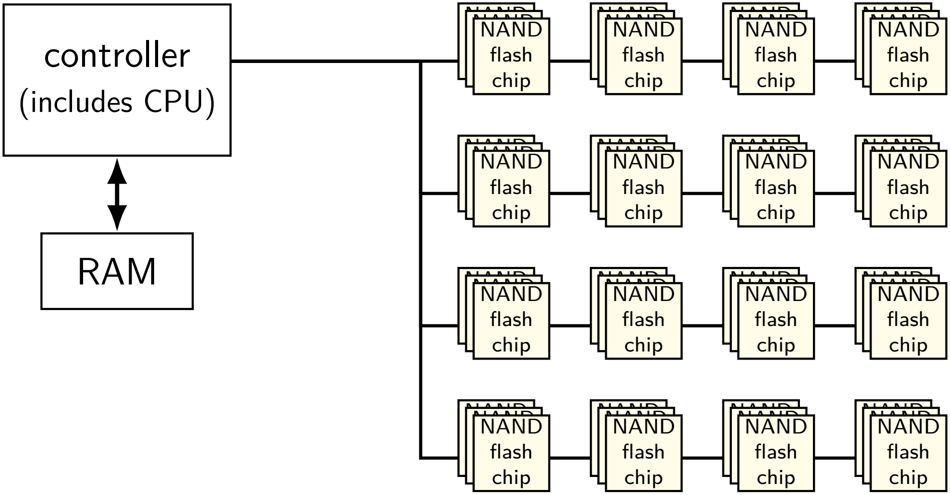
do contiguous or nearby reads/writes

- bunch of consecutive sectors much faster to read
- nearby sectors have lower seek/rotational delay

start a lot of reads/writes at once

- avoid reading something to find out what to read next
- array of sectors better than linked list

solid state disk architecture



flash

no moving parts

- no seek time, rotational latency

can read in sector-like sizes (“pages”) (e.g. 4KB or 16KB)

write once between erasures

erasure only in large *erasure blocks* (often 256KB to megabytes!)

can only rewrite blocks order tens of thousands of times

- after that, flash starts failing

SSDs: flash as disk

SSDs: implement hard disk interface for NAND flash

- read/write sectors at a time
- sectors much smaller than erasure blocks
- sectors sometimes smaller than flash 'pages'
- read/write with use sector numbers, not addresses
- queue of read/writes

need to hide erasure blocks

- trick: block remapping — move where sectors are in flash

need to hide limit on number of erases

- trick: wear levening — spread writes out

block remapping

Flash Translation Layer

remapping table

OS sector numbers

logical	physical
0	93
1	260
…	…
31	74
32	75
…	…

flash locations

block remapping

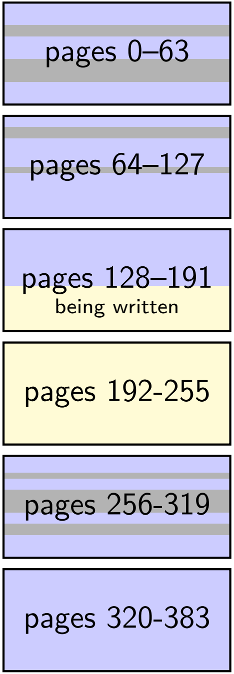
Flash
Translation
Layer
remapping table

logical	physical
0	93
1	260
...	...
31	74
32	75
...	...

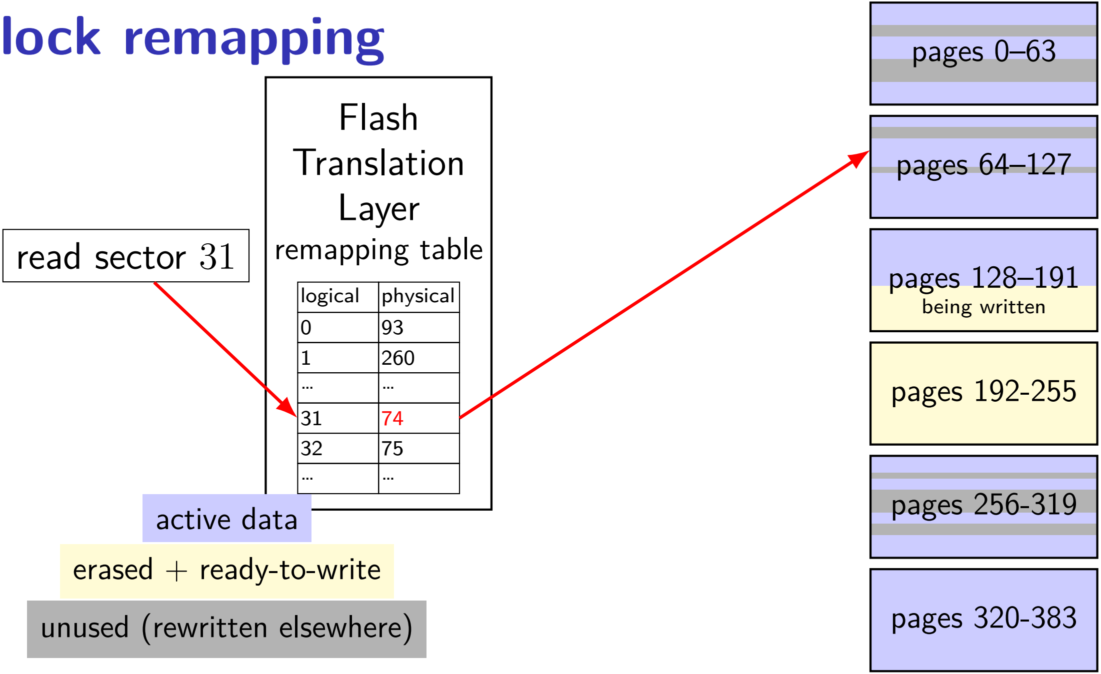
active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping



logical	physical
0	93
1	260
...	...
31	74
32	75
...	...

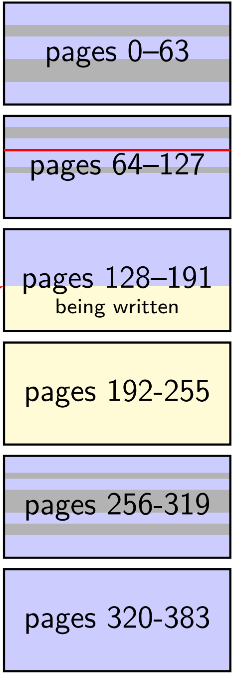
block remapping

write sector 32

Flash Translation Layer

remapping table

logical	physical
0	93
1	260
...	...
31	74
32	~~75~~ 163
...	...



active data

erased + ready-to-write

unused (rewritten elsewhere)

block remapping

Flash Translation Layer

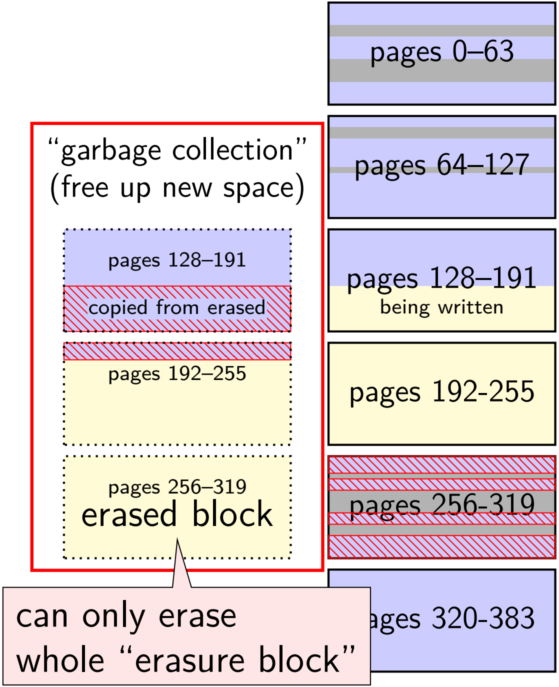
remapping table

logical	physical
0	93
1	~~260~~ 187
...	...
31	74
32	~~75~~ 163
...	...

active data

erased + ready-to-write

unused (rewritten elsewhere)



block remapping

controller contains mapping: sector $\rightarrow$ location in flash

on write: write sector to *new location*

eventually do *garbage collection* of sectors

- if erasure block contains some replaced sectors and some current sectors…
- copy current blocks to new locationt to reclaim space from replaced sectors

doing this efficiently is very complicated

SSDs sometimes have a 'real' processor for this purpose

exercise

Assuming a FAT-like filesystem on an SSD, which of the following are likely to be stored in the same (or very small number of) erasure block?

[a] the clusters of a set of log file all in one directory written continuously over months by a server and assigned a contiguous range of cluster numbers

[b] the data clusters of a set of images, copied all at once from a camera and assigned a variety of cluster numbers

[c] all the entires of the FAT (assume the OS only rewrites a sector of the FAT if it is changed)

SSD performance

reads/writes: sub-millisecond

contiguous blocks don't really matter

can depend a lot on the controller

- faster/slower ways to handle block remapping

writing can be slower, especially when almost full

- controller may need to move data around to free up erasure blocks
- erasing an erasure block is pretty slow (milliseconds?)

extra SSD operations

SSDs sometimes implement non-HDD operations

on operation: TRIM

way for OS to mark sectors as unused/erase them

SSD can remove sectors from block map

- more efficient than zeroing blocks
- frees up more space for writing new blocks

aside: future storage

emerging non-volatile memories...

slower than DRAM ("normal memory")

faster than SSDs

read/write interface like DRAM but persistent

capacities similar to/larger than DRAM